BẢNG BĂM (HASH TABLE)

DATA STRUCTURES AND ALGORITHMS

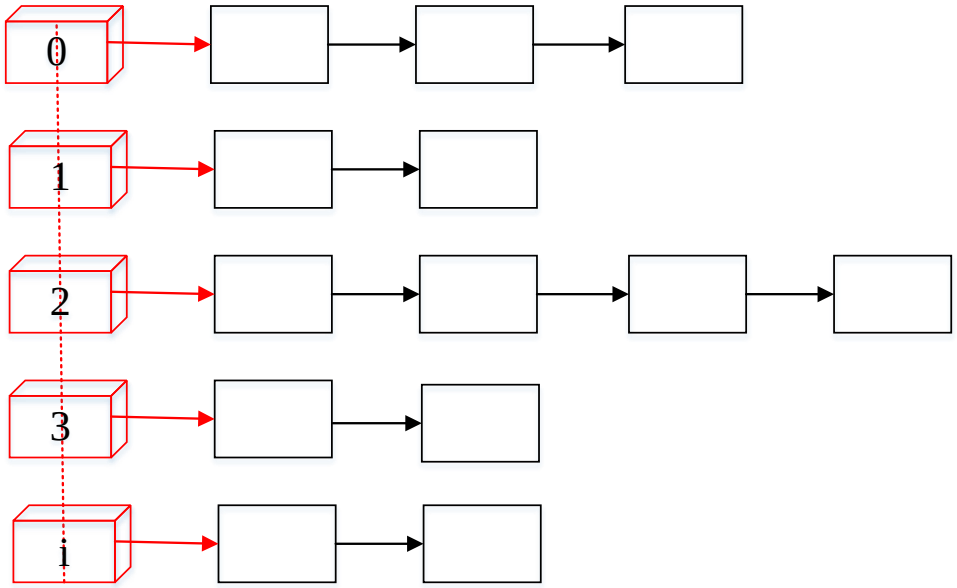
ThS Nguyễn Thị Ngọc Diễm
diemntn@uit.edu.vn

Nội dung

- Giới thiệu về bảng băm
- Hàm băm
- Sự đụng độ
- **Các phương pháp giải quyết đụng độ**
  - **Nối kết trực tiếp - Direct Chaining**
  - **Nối kết hợp nhất - Coalesced Chaining**
  - **Dò tuyến tính - Linear probing**
  - **Dò bậc hai - Quadratic probing**
  - **Băm kép - Double hashing**

Direct Chaining Method - PP nối kết trực tiếp

- Các nút bị băm cùng địa chỉ (các nút bị xung đột) được gom thành một danh sách liên kết.
- Các nút trên bảng băm được *băm* thành các danh sách liên kết. Các nút bị xung đột tại địa chỉ i được nối kết trực tiếp với nhau qua danh sách liên kết i.



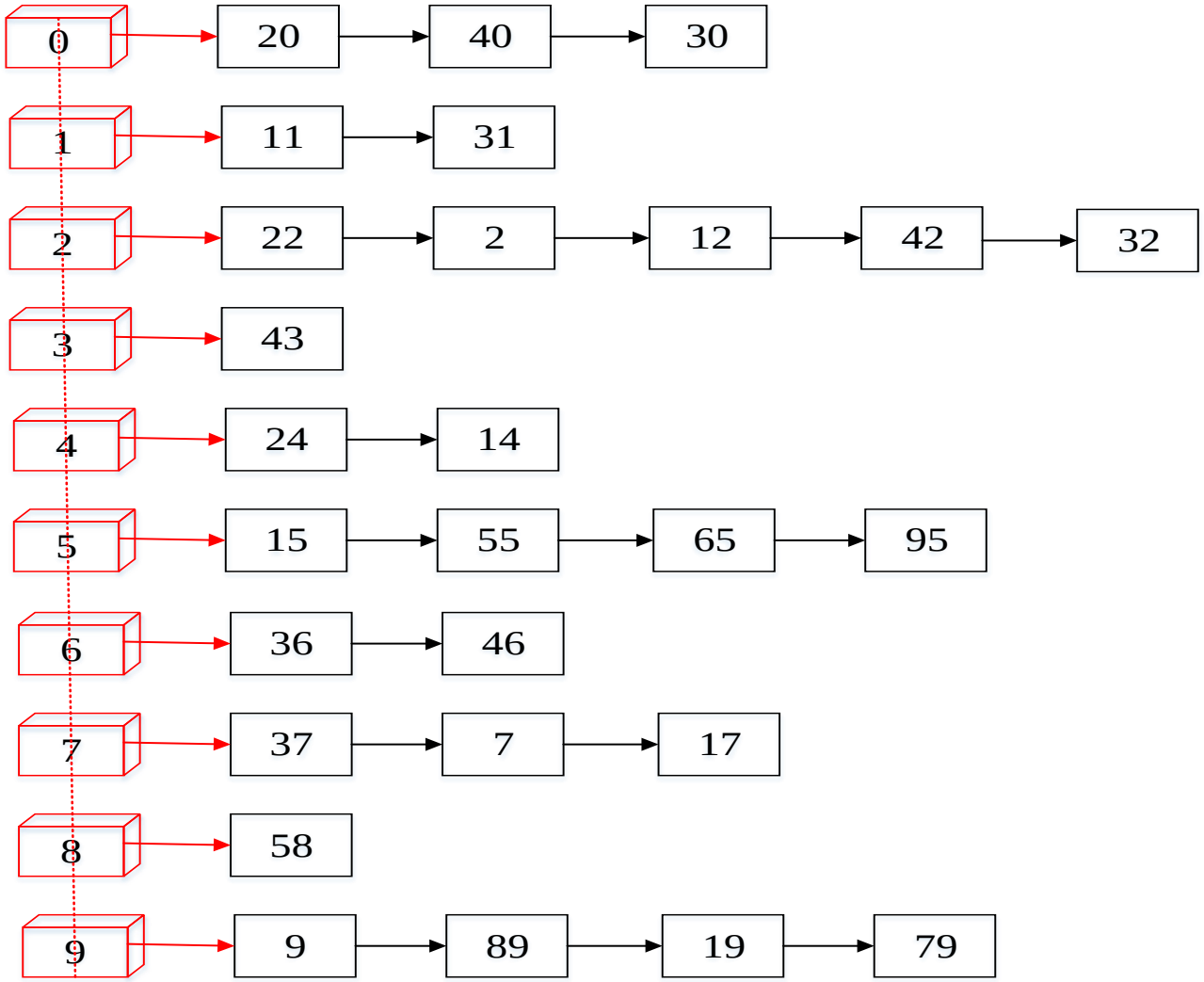
Direct Chaining Method - PP nối kết trực tiếp

- *Khi thêm một phần tử* có khóa k vào bảng băm, hàm băm h(k) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1 ứng với danh sách liên kết i mà phần tử này sẽ được thêm vào.
- *Khi tìm một phần tử* có khóa k vào bảng băm, hàm băm h(k) cũng sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1 ứng với danh sách liên kết i có thể chứa phần tử này. Như vậy, việc tìm kiếm phần tử trên bảng băm sẽ được qui về bài toán tìm kiếm một phần tử trên danh sách liên kết.

Direct Chaining Method: Ví dụ minh họa

- Xét bảng băm có cấu trúc như sau:
  - Tập khóa K: tập số tự nhiên
  - Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9}
  - Hàm băm h(key) = key % 10.
- Hình trên minh họa bảng băm vừa mô tả. Theo hình vẽ, bảng băm đã "băm" phần tử trong tập khoá K theo 10 danh sách liên kết khác nhau, mỗi danh sách liên kết gọi là một bucket:
  - Bucket 0 gồm những phần tử có khóa tận cùng bằng 0.
  - Bucket i(i=0 | … | 9) gồm những phần tử có khóa tận cùng bằng i. Để giúp việc truy xuất bảng băm dễ dàng, các phần tử trên các bucket cần thiết được tổ chức theo một thứ tự, chẳng hạn từ nhỏ đến lớn theo khóa.
  - Khi khởi động bảng băm, con trỏ đầu của các bucket là NULL.
- Theo cấu trúc này, với tác vụ insert, hàm băm sẽ được dùng để tính địa chỉ của khoá k của phần tử cần chèn, tức là xác định được bucket chứa phần tử và đặt phần tử cần chèn vào bucket này.
- Với tác vụ search, hàm băm sẽ được dùng để tính địa chỉ và tìm phần tử trên bucket tương ứng.

Direct Chaining Method: Ví dụ minh họa



(Chú ý: Khi làm bài thi cần trình bày rõ vì sao thêm khóa k vào vị trí nào i trong bảng băm)

Direct Chaining Method: Nhận xét

- Bảng băm dùng phương pháp nối kết trực tiếp sẽ ”băm” n nút vào M danh sách liên kết (M buckets).
- Tốc độ truy xuất phụ thuộc vào việc lựa chọn hàm băm sao cho băm đều n nút của bảng băm cho M buckets.
- Nếu chọn M càng lớn thì tốc độ thực hiện các phép toán trên bảng băm càng nhanh tuy nhiên không hiệu quả về bộ nhớ. Chúng ta có thể điều chỉnh M để dung hòa giữa tốc độ truy xuất và dung lượng bộ nhớ.
- Nếu chọn M=n thời gian truy xuất tương đương với truy xất trên mảng (có bậc O(1)), song tốn bộ nhớ.
- Nếu chọn M = n /k (k =2, 3, 4, …) thì ít tốn bộ nhớ hơn k lần, nhưng tốc độ chậm đi k lần.

Direct Chaining Method: Nhận xét (tt)

	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete
Sorted array	logN	N	N	logN	N/2	N/2
Unsorted array	N	N	N	N/2	N	N/2
Direct Chaining	N	N	N	1*	1*	1*

*assumes hash function is random

Direct Chaining Method: Cài đặt

```
#define M 100
struct NODE {
    int key;
    NODE *pNext;
};

// Khai báo kiểu con trỏ chỉ node
typedef NODE *NODEPTR;

/* Khai báo mảng HASHTABLE chứa M con trỏ đầu của
HASHTABLE */
NODEPTR HASHTABLE[M];
```

Direct Chaining Method: Cài đặt

Hàm băm: Giả sử chúng ta chọn hàm băm h(key) = key % M.

```
int HF(int key) {
    return key % M;
}
```

Phép toán khởi tạo bảng băm: Khởi động các HASHTABLE.

```
void InitHASHTABLE() {
   for (int i = 0; i<M; i++)
        HASHTABLE[i] = NULL;
}
```

Direct Chaining Method: Cài đặt (tt)

Phép toán kiểm tra bucket thứ i rỗng:

```
bool isEmpty(int i) {
    return(HASHTABLE[i] == NULL ? 1 : 0);
}
```

Phép toán toàn bộ bảng băm rỗng:

```
bool isEmpty() {
   for (int i = 0; i<M; i++)
        if (HASHTABLE[i] != NULL)
            return 0;
   return 1;
}
```

Direct Chaining Method: Cài đặt (tt)

Phép toán Insert: Thêm phần tử có khóa k vào bảng băm: Giả sử các phần tử trên các HASHTABLE là có thứ tự để thêm một phần tử khóa k vào bảng băm trước tiên chúng ta xác định HASHTABLE phù hợp, sau đó dùng phép toán InsertList của danh sách liên kết để đặt phần tử vào vi trí phù hợp trên HASHTABLE.

```
void Insert(int k) {
    int i = HF(k);

    // phép toán thêm khoá k vào danh sách liên kết HASHTABLE[i]
    InsertList(HASHTABLE[i], k);
}
```

Direct Chaining Method: Cài đặt (tt)

Phép toán loại bỏ: Xóa phần tử có khóa k trong bảng băm. Giả sử các phần tử trên các bucket là có thứ tự, để xóa một phần tử khóa k trong bảng băm cần thực hiện:

- Xác định bucket phù hợp
- Tìm phần tử để xóa trong bucket đã được xác định, nếu tìm thấy phần tử cần xóa thì loại bỏ phần tử theo các phép toán tương tự loại bỏ một phần tử trong danh sách liên kết.

Direct Chaining Method: Cài đặt (tt)

Phép toán loại bỏ (tt)

```
void Remove(int k) {
    NODEPTR Q, p;
    int i = HF(k);
    p = HASHTABLE[i];
    Q = p;
    while (p != NULL &&  p->key != k) {
        Q = p;
        p = p->pNext;
    }
    if (p == NULL) cout <<"\n Không có nút có khóa " << k;
    else if (p == HASHTABLE[i])
            pop(i);
    else
            delAfterQ(i, q, Q);
}
```

Direct Chaining Method: Cài đặt (tt)

- **Phép toán tìm kiếm:**

```
NODEPTR Search(int k){
    int i = HF(k);
    NODEPTR p = HASHTABLE[i];

    while (p != NULL  && k != p->key)
          p = p->pNext;

    return p;
}
```

Direct Chaining Method: Cài đặt (tt)

Phép toán duyệt HASHTABLE[i]:

Duyệt các phần tử trong HASHTABLE thứ i:

```
void traverseHASHTABLE(int i) {
    NODEPTR p = HASHTABLE[i];
    while (p != NULL) {
        cout << p->key << "\t";
        p = p->pNext;
    }
}
```

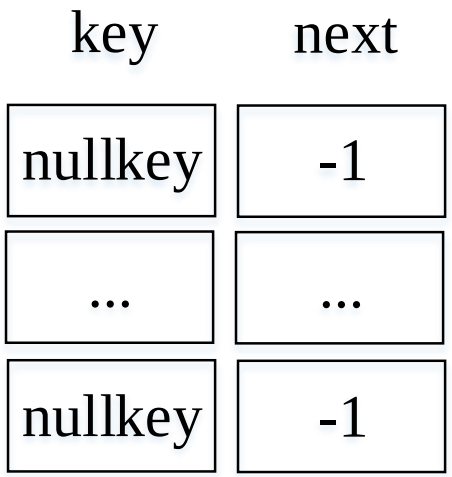
Direct Chaining Method: Cài đặt (tt)

Phép toán duyệt toàn bộ bảng băm:

Duyệt toàn bộ bảng băm.

```
void traverse() {
    for (int i = 0; i<M; i++) {
        cout << endl << "Butket " << i << ": ";
        traverseHASHTABLE(i);
    }
}
```

Coalesced Chaining Method - PP nối kết hợp nhất



Coalesced Chaining

- Bảng băm trong trường hợp này được cài đặt bằng danh sách liên kết dùng mảng, có M nút. Các nút bị xung đột địa chỉ được nối kết nhau qua một danh sách liên kết.
- Mỗi nút của bảng băm là một mẫu tin có 2 trường:
  - Trường key: chứa các khóa node
  - Trường next: con trỏ chỉ node kế tiếp nếu có xung đột.
- Khi khởi động bảng băm thì tất cả trường key được gán nullkey, tất cả trường next được gán -1.

Coalesced Chaining

- Khi thêm một nút có khóa key vào bảng băm, hàm băm H(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1.
- Nếu chưa bị xung đột thì thêm nút mới vào địa chỉ này .
- Nếu bị xung đột thì nút mới được cấp phát là nút trống phía cuối mảng. Cập nhật liên kết next sao cho các nút bị xung đột hình thành một danh sách liên kết.
- Khi tìm một nút có khóa key trong bảng băm, hàm băm H(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1, tìm nút khóa key trong danh sách liên kết xuất phát từ địa chỉ i.

Coalesced Chaining Method: Ví dụ 1

- Minh họa cho bảng băm có tập khóa là tập số tự nhiên, tập địa chỉ có 10 địa chỉ (M=10) (từ địa chỉ 0 đến 9), chọn hàm băm h(key) = key mod 10.
- Tập keys = {30, 24, 26, 10, 14, 54, 4}

(Chú ý: Khi làm bài thi cần trình bày rõ vì sao thêm khóa k vào vị trí nào i trong bảng băm)

0	30	9
1		-1
2		-1
3		-1
4	24	8
5	4	-1
6	26	-1
7	54	5
8	14	7
9	10	-1

Coalesced Chaining Method: Ví dụ 2

- Minh họa cho bảng băm có tập khóa là tập số tự nhiên, tập địa chỉ có 10 địa chỉ (M=10) (từ địa chỉ 0 đến 9), chọn hàm băm h(key) = key mod 10.
- Tập keys = {30, 24, 26, 10, 14, 54, 4, 8, 84}

(Chú ý: Khi làm bài thi cần trình bày rõ vì sao thêm khóa k vào vị trí nào i trong bảng băm)

0	30	9
1		-1
2	84	-1
3	8	2
4	24	8
5	4	3
6	26	-1
7	54	5
8	14	7
9	10	-1

Coalesced Chaining Method: Ví dụ 3

- Minh họa cho bảng băm có tập khóa là tập số tự nhiên, tập địa chỉ có 10 địa chỉ (M=10) (từ địa chỉ 0 đến 9), chọn hàm băm h(key) = key mod 10.
- Tập keys = {20, 31, 10, 51, 84, 50, 1, 24, 90}

(Chú ý: Khi làm bài thi cần trình bày rõ vì sao thêm khóa k vào vị trí nào i trong bảng băm)

0	20	9
1	31	8
2	nullkey	-1
3	90	-1
4	84	5
5	24	-1
6	1	-1
7	50	3
8	51	6
9	10	7

Coalesced Chaining Method: Nhận xét

- Thực chất cấu trúc bảng băm này chỉ tối ưu khi băm đều, nghĩa là mỗi danh sách liên kết chứa một vài phần tử bị xung đột, tốc độ truy xuất lúc này có bậc O(1).
- Trường hợp xấu nhất là băm không đều vì hình thành một danh sách có n phần tử nên tốc độ truy xuất lúc này có bậc O(n).

Coalesced Chaining Method: Cài đặt

Chương trình cài đặt bằng danh sách kề

- **Khai báo cấu trúc bảng băm:**

```
#define nullkey -1

#define M 100

// Khai báo cấu trúc một node trong bảng băm

struct NODE{
    int key;
    int next; // con trỏ chỉ nút kế tiếp khi có xung đột

};

// Khai báo bảng băm

NODE HASHTABLE[M];

int avail; // biến toàn cục chỉ nút trống ở cuối bảng
băm được cập nhật khi có xung đột
```

Coalesced Chaining Method: Cài đặt (tt)

- **Hàm băm:**

```
// Giả sử chúng ta chọn hàm băm dạng mod: h(key) = key % 10.
Có thể dùng một hàm băm bất kì thay cho hàm băm dạng % trên.

int HF(int key) {
    return key % 10;

}
```

- **Phép toán khởi tạo:**

```
void Initialize() {
    for (int i = 0; i<M; i++){
        HASHTABLE[i].key = nullkey; HASHTABLE[i].next = -1;
    }
    avail = M - 1; // nút M-1 là nút ở cuối bảng chuẩn bị cấp
    phát nếu có xung đột

}
```

Coalesced Chaining Method: Cài đặt (tt)

- **Phép toán tìm kiếm:**

```
int Search(int k) {
    int i = HF(k);
    while (k != HASHTABLE[i].key && i != -1)
          i = HASHTABLE[i].next;

    if (k == HASHTABLE[i].key)
          return i; //tìm thấy

    return M; //không tìm thấy
}
```

Coalesced Chaining Method: Cài đặt (tt)

- **Phép toán lấy phần tử trống cuối bảng băm:**

Chọn phần tử còn trống phía cuối bảng băm để cấp phát khi xảy ra xung đột.

```
int getEmpty() {
   while (HASHTABLE[avail].key != nullkey)
         avail --;
   return avail;
}
```

Coalesced Chaining Method: Cài đặt (tt)

- **Phép toán Insert**

```
int Insert(int k) {
    int i = Search(k), j; // j là địa chỉ nút trong được cấp phát
    if (i != M) {
      cout << "\n Khoa " << k << " bi trung, khong them nut nay duoc!";
      return i;
    }
    i = HF(k);
    while (HASHTABLE[i].next >= 0) i = HASHTABLE[i].next;
    if (HASHTABLE[i].key == nullkey) j = i; // Nút i còn trống thì cập nhật
    else { // Nếu nút i là nút cuối của danh sách
       j = getEmpty();
       if (j < 0) { cout << "\n Bang bam bi day!“; return j; }
       HASHTABLE[i].next = j;
    }
    HASHTABLE[j].key = k;
    return j;
}
```

Open addressing

H(key, i) = (h(key) + f(i)) % TableSize

Trong đó:

- *key: từ khóa cần băm*
- *h(key): hàm băm chính.*
- *H(key, f(i)): hàm băm lại lần thứ i.*
- *TableSize: kích thước bảng băm.*
- *%: phép lấy dư (mod)*
- *f(i): hàm thể hiện của các phương pháp xử lý đụng độ: dò tuyến tính, dò bậc hai, băm kép, f(0)=0*

* Phương pháp này không sử dụng con trỏ

Open addressing (tt)

$$H(key, i) = (h(key) + f(i)) \% TableSize$$

- Với phương pháp dò tuyến tính thì f(i) = i

$$H(key, i) = (h(key) + i) \% TableSize$$

- Với phương pháp dò bậc 2 thì $f(i) = i^2$

$$H(key, i) = (h(key) + i^2) \% TableSize$$

- Với phương pháp băm kép thì f(i) = i * h2(key)

$$H(key, i) = (h1(key) + i * h2(key)) \% TableSize$$

Linear Probing Method - PP dò tuyến tính

- Hàm băm lại của phương pháp dò tuyến tính là truy xuất địa chỉ kế tiếp. Hàm băm lại lần i được biểu diễn bằng công thức sau:

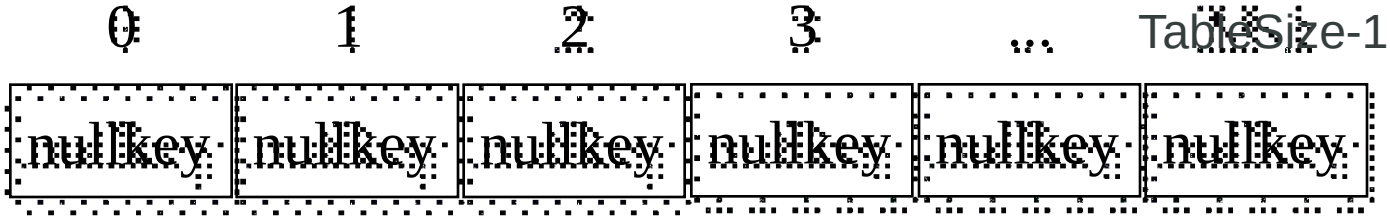
$$H(key, i) = (h(key) + i) \bmod TableSize$$

với h(key) là hàm băm chính của bảng băm, f(i)=i

- Lưu ý địa chỉ dò tìm kế tiếp là địa chỉ 0 nếu đã dò đến cuối bảng.

Linear Probing Method

- Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có TableSize nút, mỗi nút của bảng băm là một mẫu tin có một trường key để chứa khóa của nút. So at any point, size of the table must be greater than or equal to the total number of keys
- Khi khởi động bảng băm thì tất cả trường key được gán nullkey.
- Khi thêm nút có khoá key vào bảng băm, hàm băm h(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến TableSize-1:
  - Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ này.
  - Nếu bị xung đột thì hàm băm lại lần 1, hàm h_1 sẽ xét địa chỉ kế tiếp, nếu lại bị xung đột thì hàm băm thì hàm băm lại lần 2, hàm h_2 sẽ xét địa chỉ kế tiếp nữa, …, và quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử mới vào địa chỉ này.



Linear Probing Method

- Nếu chưa bị xung đột thì thêm nút mới vào địa chỉ này.
- Nếu bị xung đột thì hàm băm lại lần 1 (H(key, 1)) sẽ xét địa chỉ kế tiếp, nếu lại bị xung đột thì hàm băm lại lần 2 (H(key, 2)) sẽ xét địa chỉ kế tiếp …
- Quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm nút vào địa chỉ này.
- Khi tìm một nút có khoá key vào bảng băm, hàm băm h(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến TableSize-1, tìm nút khoá key trong khối đặc chứa các nút xuất phát từ địa chỉ i.

Linear Probing Method: Ví dụ

- Giả sử, khảo sát bảng băm có cấu trúc như sau:
  - Tập khóa K: tập số tự nhiên
  - Tập địa chỉ TableSize: gồm 10 địa chỉ (TableSize ={0, 1, …, 9})
  - Hàm băm h(key) = key mod 10.
- Hình thể hiện thêm các giá trị 41, 45, 51, 30, 62, 80, 53, 89, 77, 19 vào bảng băm.

Linear Probing Method: Ví dụ (tt)

- Thêm: {41, 45, 51, 30, 62, 80, 53, 89, 77, 19}
- H(key, i) = (h(key) + i) % TableSize (Chú ý: Khi làm bài thi cần trình bày rõ vì sao thêm khóa k vào vị trí nào i trong bảng băm)

	Empty Table	Thêm 41	Thêm 45	Thêm 51	Thêm 30	Thêm 62	Thêm 80	Thêm 53	Thêm 89	Thêm 77	Thêm 19
0					30	30	30	30	30	30	30
1		41	41	41	41	41	41	41	41	41	41
2				51	51	51	51	51	51	51	51
3						62	62	62	62	62	62
4							80	80	80	80	80
5			45	45	45	45	45	45	45	45	-1
6								53	53	53	53
7										77	77
8											19
9									89	89	89

Linear Probing Method: Nhận xét

- Bảng băm này chỉ tối ưu khi băm đều, nghĩa là trên bảng băm các khối đặc chứa vài phần tử và các khối phần tử chưa sử dụng xen kẽ nhau, tốc độ truy xuất lúc này có bậc O(1).
- Trường hợp xấu nhất là băm không đều hoặc bảng băm đầy, lúc này hình thành một khối đặc có n phần tử, nên tốc độ truy xuất lúc này có bậc O(n).

Linear Probing Method: Nhận xét (tt)

	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete
Sorted array	logN	N	N	logN	N/2	N/2
Unsorted array	N	N	N	N/2	N	N/2
Direct Chaining	N	N	N	1*	1*	1*
Linear Probing	N	N	N	1*	1*	1*

*assumes hash function is random

Linear Probing Method: Cài đặt

```
#define nullkey -1
#define MAXTABLESIZE 10000
struct NODE { int key; //khóa của node trên bảng băm };
typedef NODE HASHTABLE[MAXTABLESIZE];
int TableSize;
int main() {
     HASHTABLE H; //or: NODE H[MAXTABLESIZE];
     int CurrentSize, k;
     cin >> TableSize;
     CreateHashTable(H, CurrentSize);
     Traverse(H, CurrentSize);
      cin >> k;
     if (Search(H, CurrentSize, k) != -1)
          cout << "Found " << k << ".";
     else cout << "Not Found " << k << ".";
     return 0;
}
```

Linear Probing Method: Cài đặt (tt)

Hàm băm:

Giả sử chúng ta chọn hàm băm dạng %: H(key) = key % 10.

```
int HF(int key) { return key%M; }
int HF_LinearProbing(int key, int i) {
   return (HF(key) + i) % TableSize;
}
```

Chúng ta có thể dùng một hàm băm bất kì thay cho hàm băm dạng % trên.

Phép toán khởi tạo:

Gán tất cả các phần tử trên bảng có trường key là nullkey. Gán biến toàn cục N=0.

```
void Initialize(HASHTABLE &H, int &CurrentSize) {
   for (int i = 0; i<TableSize; i++)
      H[i].key =  nullkey;
   CurrentSize = 0; //số node hiện có khởi động bằng 0
}
```

Linear Probing Method: Cài đặt

Phép toán kiểm tra trống:

Kiểm tra bảng băm có trống hay không.

```
int isEmpty(const HASHTABLE &H, int CurrentSize) {
   return CurrentSize == 0 ? true : false;
}
```

Phép toán kiểm tra đầy:

Kiểm tra bảng băm đã đầy chưa.

```
int isFull(const HASHTABLE &H, int CurrentSize) {
   return CurrentSize == TableSize-1 ? true : false;
}
```

Lưu ý bảng băm đầy khi N=M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm để thuận tiện trong quá trình xử lý.

Linear Probing Method: Cài đặt (tt)

Phép toán Search: // Việc tìm kiếm phần tử có khoá k trên một khối đặc, bắt đầu từ một địa chỉ i = HF(k), nếu không tìm thấy phần tử có khoá k, hàm này sẽ trả về trị -1, còn nếu tìm thấy, hàm này trả về địa chỉ tìm thấy.

```
int Search(const HASHTABLE &H, const int &CurrentSize, int k) {
   int b = HF(k), i=0;
   while ( H[b].key != k && H[b].key !=  nullkey) {
      b=HF_LinearProbing(k, ++i);
   }
   if (H[b].key == k) //tìm thấy
      return b;
   return -1; // không tìm thấy
}
```

Linear Probing Method: Cài đặt (tt)

Phép toán Insert: `// Thêm phần tử có khoá k vào bảng băm.`

```
int Insert(HASHTABLE &H, int &CurrentSize, int k) {
   if (isFull(H, CurrentSize)) return -2; // HashTable đầy
   int b = HF(k), i=0;
   while ( b<TableSize && H[b].key != k && H[b].key !=  nullkey) {
      b=HF_LinearProbing(k, ++i);
   }

   if (H[b].key == k)
      return -1; // giá trị k tồn tại trong mảng

   H[b].key = k;
   CurrentSize += 1;
   return b;
}
```

Linear Probing Method: Cài đặt (tt)

Phép toán Remove: // xóa node tại vị trí i trên bảng băm.

```
void Remove(int i) {
     int j, r, a, cont=true;
     do {
          HashTable[i].key = nullkey;
          j = i;
          do {
            i = i + 1;
            if (i >= M) i = i- M;
            if (HashTable[i].key == nullkey) cont = false;
            else {
              r = HF(HashTable[i].key);
              a = (j <r && r <= i) || (r <= i && i <j) || (i<j && j <r);
            }
            } while (cont && a);
          if (cont) HashTable[j].key = HashTable[i].key;
     } while (cont);

}
```

Quadratic Probing Method - PP dò bậc hai

- Hàm băm lại của phương pháp dò bậc hai là truy xuất các địa chỉ cách bậc 2. Hàm băm lại hàm i được biểu diễn bằng công thức sau:

$$H(key, i) = (h(key) + i^2) \% M$$

với h(key) là hàm băm chính của bảng băm, $f(i) = i^2$

- Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.
- Bảng băm với phương pháp do bậc hai nên chọn số địa chỉ M là số nguyên tố.

Quadratic Probing Method

- Bảng băm dùng phương pháp dò tuyến tính bị hạn chế do rải các nút không đều, bảng băm với phương pháp dò bậc hai rải các nút đều hơn.
- Bảng băm trong trường hợp này được cài đặt bằng danh sách kế có M nút, mỗi nút của bảng băm là một mẩu tin có một trường key để chứa khóa các nút.
- Khi khởi động bảng băm thì tất cả trường key bị gán nullkey.
- Khi thêm nút có khóa key vào bảng băm, hàm băm h(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1.

Quadratic Probing Method

- Nếu chưa bị xung đột thì thêm nút mới vào địa chỉ i.
- Nếu bị xung đột thì hàm băm lại lần 1 H(key, 1) sẽ xét địa chỉ cách 1^2, nếu lại bị xung đột thì hàm băm lại lần 2 H(key, 2) sẽ xét địa chỉ cách i 2^2,..., quá trình cứ thế cho đến khi nào tìm được trống và thêm nút vào địa chỉ này.
- Khi tìm môt nút có khóa key trong bảng băm thì xét nút tại địa chỉ i=h(key), nếu chưa tìm thấy thì xét nút cách i 1^2, 2^2, ..., quá trình cứ thế cho đến khi tìm được khóa (trường hợp tìm thấy) hoặc rơi vào địa chỉ trống (trường hợp không tìm thấy).

Quadratic Probing Method: Ví dụ

- Giả sử, khảo sát bảng băm có cấu trúc như sau:
  - Tập khóa K: tập số tự nhiên
  - Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9})
  - Hàm băm h(key) = key % 10.
- Hình thể hiện thêm các giá trị 10, 15, 16, 20, 30, 25, 26, 36 vào bảng băm.

Quadratic Probing Method: Ví dụ (tt)

Thêm vào các khóa 10, 15, 16, 20, 30, 25, 26, 36:

$$M=10,\ h(key)= key\%10,\ H(key, i) = (h(key) + i^2)\ \%\ 10$$

(Chú ý: Khi làm bài thi cần trình bày rõ vì sao thêm khóa k vào vị trí nào i trong bảng băm)

	Empty Table	**Thêm 10**	**Thêm 15**	**Thêm 16**	**Thêm 20**	**Thêm 30**	**Thêm 25**	**Thêm 26**	**Thêm 36**
0		10	10	10	10	10	10	10	10
1					20	20	20	20	20
2									36
3									
4						30	30	30	30
5			15	15	15	15	15	15	15
6				16	16	16	16	16	16
7								26	26
8									
9							25	25	25

Quadratic Probing Method: Nhận xét

- Nên chọn số địa chỉ M là số nguyên tố. Khi khởi động bảng băm thì tất cả M trường key được gán nullkey, biến toàn cục N được gán 0.
- Bảng băm đầy khi N = M-1, và nên dành ít nhất một phần tử trống trên bảng băm.
- Bảng băm này tối ưu hơn bảng băm dùng phương pháp dò tuyến tính do rải rác phần tử đều hơn, nếu bảng băm chưa đầy thì tốc độ truy xuất có bậc O(1). Trường hợp xấu nhất là bảng băm đầy vì lúc đó tốc độ truy xuất chậm do phải thực hiện nhiều lần so sánh.

Quadratic Probing Method: Cài đặt

```
#define nullkey -1
#define M 97

// Khai báo node của bảng băm
struct NODE {
    int key; // Khóa của node trên bảng băm
};

// Khai báo bảng băm có M node
NODE HASHTABLE[M];

// Biến toàn cục chỉ số nút hiện có trên bảng băm
int  N;
```

Quadratic Probing Method: Cài đặt (tt)

Hàm băm: Giả sử chúng ta chọn hàm băm dạng%: f(key)=key%10.

```
int HF(int key) {
    return key % 10;
}
```

Phép toán khởi tạo: Gán tất cả các phần tử trên bảng có trường key là nullkey. Gán biến toàn cục N=0.

```
void Initialize() {
   for (int i = 0; i<M; i++)
         HASHTABLE[i].key = nullkey;
   N = 0; // Số node hiện có ban đầu bằng 0
}
```

Quadratic Probing Method: Cài đặt (tt)

Phép toán kiểm tra trống:

Kiểm tra bảng băm có trống hay không.

```
int isEmpty() {
     return N == 0 ? true : false;
}
```

Phép toán kiểm tra đầy:

Kiểm tra bảng băm đã đầy chưa.

```
int isFull() {
     return N == M - 1 ? true : false;
}
```

Lưu ý bảng băm đầy khi N=M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm.

Quadratic Probing Method: Cài đặt (tt)

Phép toán tìm kiếm: Tìm phần tử có khóa k trên bảng băm,nếu không tìm thấy hàm này trả về trị M, nếu tìm thấy hàm này trả về địa chỉ tìm thấy.

```
int Search(int k) {
    int i = HF(k), d = 1;
    while (HASHTABLE[i].key != k && HASHTABLE[i].key !=
    nullkey) {
     // Băm lại theo phương pháp dò bậc hai
        i = (i + d) % M;
        d = d + 2;
    }
    if (HASHTABLE[i].key == k)
            return i;
    return M;
}
```

Quadratic Probing Method: Cài đặt (tt)

Phép toán Insert: Thêm phần tử có khoá k vào bảng băm.

```
int Insert(int k) {
    int i , d;
    if (Search(k)<M)
        return M; // Trùng khoá
    if (isFull()) {
        cout << “Full!";
        return;
    }

    i = HF(k);
    d = 1;
    while (HASHTABLE[i].key !=
    nullkey) {
        i = (i + d) % M;
        d = d + 2;
    }

    HASHTABLE[i].key = k;
    N = N + 1;
    return i;
}
```

Double Hash Method - PP Băm kép

- Hàm băm lại lần i được biểu diễn bằng công thức sau:

$$H(key, i) = (h1(key) + i * h2(key)) \% M$$

với h1(key) là hàm băm chính của bảng băm, f(i)= i* h2(key)

- Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.
- Bảng băm với phương pháp băm kép nên chọn số địa chỉ M là số nguyên tố.

Double Probing Method - PP Băm kép

- Bảng băm này dùng hai hàm băm khác nhau với mục đích để rải rác đều các phần tử trên bảng băm.
- Chúng ta có thể dùng hai hàm băm bất kỳ, ví dụ chọn hai hàm băm như sau:

$$h1(key) = key \% M$$

$$h2(key) = R - key \% R$$ (thường chọn R là số nguyên tố nhỏ hơn M)

- Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử, mỗi phần tử của bảng băm là một mẫu tin có một trường key để lưu khoá các phần tử.

Double Probing Method

- **Khi khởi động:** bảng băm, tất cả trường key được gán nullkey.
- **Khi thêm phần tử:** có khoá key vào bảng băm, thì **i=h1(key)** và **j=h2(key)** sẽ xác định địa chỉ i và j trong khoảng từ 0 đến M-1:
  - Nếu chưa bị xung đột thì thêm phần tử mới tại địa chỉ i này.
  - Nếu bị xung đột thì hàm băm lại lần 1 f1 sẽ xét địa chỉ mới i+j, nếu lại bị xung đột thì hàm băm lại lần 2 là f2 sẽ xét địa chỉ i+2j, …, quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử vào địa chi này.

Double Probing Method: Ví dụ

Thêm vào các khóa 89, 18, 49, 58, 69:

M=10, h1(key)= key%10, h2(key) = 7 - key%7

H(key, i) = (h1(key) + i* h2(key)) % M

	Empty Table	Thêm 89	Thêm 18	Thêm 49	Thêm 58	Thêm 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

(Chú ý: Khi làm bài thi cần trình bày rõ vì sao thêm khóa k vào vị trí nào i trong bảng băm)

Double Probing Method

- **Khi tìm kiếm** một phần tử có khoá key trong bảng băm, hàm băm **i=h1(key)** và **j=h2(key)** sẽ xác định địa chỉ i và j trong khoảng từ 0 đến M-1. Xét phần tử tại địa chỉ i, nếu chưa tìm thấy thì xét tiếp phần tử i+j, i+2j, …, quá trình cứ thế cho đến khi nào tìm được khoá (trường hợp tìm thấy) hoặc bị rơi vào địa chỉ trống (trường hợp không tìm thấy).

Double Probing Method: Nhận xét

- Nên chọn số địa chỉ M là số nguyên tố.
- Bảng băm đầy khi N = M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm.
- Bảng băm được cài đặt theo cấu trúc này linh hoạt hơn bảng băm dùng phương pháp dò tuyến tính và bảng băm dùng phương pháp sò bậc hai, do dùng hai hàm băm khác nhau nên việc rải phần tử mang tính ngẫu nhiên hơn, nếu bảng băm chưa đầy tốc độ truy xuất có bậc O(1). Trường hợp xấu nhất là bảng băm gần đầy, tốc độ truy xuất chậm do thực hiện nhiều lần so sánh.

Double Probing Method: Cài đặt

```
#define nullkey -1
#define M 100
// Khai báo cấu trúc một node của bảng băm
struct NODE{
    int key; // Khóa của nút trên bảng băm
};

// Khai báo bảng băm có M nút
NODE HASHTABLE[M];

// Biến toàn cục chỉ số nút hiện có trên bảng băm
int N;
```

Double Probing Method: Cài đặt (tt)

Hàm băm: Giả sử chúng ta chọn hai hàm băm dạng %:

h1(key)=key % M và h2(key) =(M-2) – key % (M-2).

```
//Hàm băm thứ nhất
int HF(int key) {
     return key % M;
}

//Hàm băm thứ hai
int HF2(int key) {
     return M - 2 - key % (M-2);
}
```

Double Probing Method: Cài đặt (tt)

Phép toán Khởi tạo:

- Khởi động bảng băm.
- Gán tất cả các phần tử trên bảng có trường key là nullkey.
- Gán biến toàn cục N = 0.

```
void Initialize() {
   for (int i = 0; i<M; i++)
         HASHTABLE[i].key = nullkey;
   N = 0; // số nút hiện có khởi động bằng 0
}
```

Double Probing Method: Cài đặt (tt)

```
//Kiểm tra bảng băm có rỗng không
int isEmpty() {
     return N == 0 ? true : false;
}

// Kiểm tra bảng băm đã đầy chưa
int isFull() {
     return N == M - 1 ? true : false;
}
```

- Lưu ý bảng băm đầy khi N=M-1, chúng ta nên dành ít nhất một phần tử trống trên bảng băm.

Double Probing Method: Cài đặt (tt)

Phép toán tìm kiếm:

```
int Search(int k) {
    int  i, j;
    i = HF(k);
    j = HF2(k);
    while (HASHTABLE[i].key != k && HASHTABLE[i].key != nullkey)
           i = (i + j) % M; // băm lại (theo phương pháp băm kép)

    // tìm thấy
    if (HASHTABLE[i].key == k)
           return i;

    // không tìm thấy
    return M;
}
```

Double Probing Method: Cài đặt (tt)

Phép toán Insert: Thêm phần tử có khoá k vào bảng băm.

```
int Insert(int k) {
    int i, j;
    if (Search(k) < M) {
        cout << "Da co khoa nay
        trong bang bam!";
        return M;
    }

    if (isFull()) {
        cout << "Bang bam bi day!";
        return M;
    }
    i = HF(k);
    j = HF2(k);
    while (HASHTABLE[i].key !=
    nullkey)
        // Băm lại theo phương
        pháp băm kép)
        i = (i + j) % M;

    HASHTABLE[i].key = k;
    N = N + 1;
    return i;
}
```

Double Probing Method: Cài đặt (tt)

Phép toán Delete: Xóa phần tử có khoá k vào bảng băm.

Câu hỏi và Bài tập

1. Hãy cài đặt hàm băm sử dụng phương pháp nối kết trực tiếp.

2. Hãy cài đặt hàm băm sử dụng phương pháp băm kép.

3. Giả sử kích thước của bảng băm là SIZE = s và d1, d2, …, ds-1 là hoán vị ngẫu nhiên của các số 1, 2, …, s-1. Dãy thăm dò ứng với khoá k được xác định như sau:

$$i_0 = i = h(k)$$

$$i_m = (i + d_i) \% SIZE , 1 \leq m \leq s - 1$$

Hãy cài đặt hàm thăm dò theo phương pháp trên.

Câu hỏi và Bài tập

4. Cho cỡ bảng băm SIZE = 11. Từ bảng băm rỗng, sử dụng hàm băm chia lấy dư, hãy đưa lần lượt các dữ liệu với khoá:

32 , 15 , 25 , 44 , 36 , 21

vào bảng băm và đưa ra bảng băm kết quả trong các trường hợp sau:

a. Bảng băm được chỉ mở với thăm dò tuyến tính.

b. Bảng băm được chỉ mở với thăm dò bình phương.

c. Bảng băm dây chuyền.

5. Từ các bảng băm kết quả trong bài tập 4, hãy loại bỏ dữ liệu với khoá là 44 rồi sau đó xen vào dữ liệu với khoá là 65.

Chúc các em học tốt!